# Amalgamating NAF and NAI

Alessandra Di Pierro

Dipartimento di Informatica, Università di Pisa,

Corso Italia 40, 56125 Pisa, Italy

e_mail: dipierro@dipisa.di.unipi.it

**Abstract**

We define a new logic language with negation by introducing existential and universal quantifiers in the queries. We call $\Gamma-$*general* programs and $\Gamma-$*general* goals the programs and the goals in this language.

Negation is handled by using two rules: the NAF rule for universally quantified negative queries and the NAI rule for existentially quantified negative queries. We formalize this amalgamation in terms of NAIF-resolution (SLD-resolution with Negation As Instantiation and Failure), and we prove its correctness w.r.t. the completion semantics.

We also show how extended programs and goals, i.e. programs and goals whose body is an arbitrary first-order formula ([11]), can be transformed into $\Gamma-$*general* programs and goals and then evaluated using our system.

## 1 Introduction

SLD-resolution extended by the NAF rule for treating negative subgoals, (SLDNF-resolution), represents the most widely used method to compute with general logic programs and goals. In the literature, a major limitation of the SLDNF-resolution is that only ground negative literals can be treated by it; hence the computation cannot proceed when the current goal contains only negative non-ground literals (floundering). It is well-known that a sufficient condition for non-floundering is that queries are allowed, i.e. every variable in a negative literal occur also in a positive literal in the query ([4]).

On the other hand, this problem is avoided if we use the NAI rule ([6]), for solving negative (possibly non-ground) subgoals, thus obtaining SLDNI-resolution ([7]). When $\neg A$ is selected, it succeeds if there exists a finitely instantiating SLDNI-tree for $\leftarrow A$.; it fails if there exists a non-instantiating SLDNI-refutation for $\leftarrow A$. However, also SLDNI-resolution has a limitation, in that it can only deal with queries where no negative components share variables with the other components in the query.

In this sense, the two approaches are orthogonal to each other, so it is obvious to think that they could be smoothly combined.

In this paper we propose an amalgamation of SLDNF-resolution and SLDNI-resolution, which should provide the advantages of both of them and reduce their limitations. We call it NAIF-resolution, for SLD-resolution with Negation As Instantiation and Failure. In particular, the resulting system would borrow from SLDNF-resolution the possibility to infer universally quantified negative queries, which is not offered by SLDNI-resolution. On the other hand, the new interpreter would inherit from it the ability to deduce existentially quantified negative queries. Furthermore, in the amalgamation we show how non-ground negative atoms can be treated correctly.

We recall that SLDNF-resolution and SLDNI-resolution are correct, respectively, w.r.t. the Clark's completion semantics $Comp(P)$ ([4]), and w.r.t. a completion semantics, $Comp_L(P)$, based on a language $L$ with infinitely many constant symbols ([8]). However, SLDNF-resolution is correct w.r.t. $Comp_L(P)$ too, because $Comp_L(P) \models Comp(P)$ holds (Proposition 3.19 in [8]).

Since NAI and NAF have different semantics, we prefer to introduce this difference into the syntax and explicitly use $\exists \bar{x}.\neg A$ and $\forall \bar{x}.\neg A$ in the program clauses and in the goals. We further generalize this extension by allowing for queries expressions like $\exists \bar{x}.\neg Q$ and $\forall \bar{x}.\neg Q$, where $Q$ is in its turn a query. We call these $\Gamma$*−general* programs and $\Gamma$*−general* goals. The introduction of (universal or existential) quantifiers in the syntax means that quantification of negated queries is not automatically inferred from the semantics anymore, so that we can specify which variables of the query have to be intended as quantified. In this way, variables which are free in the negative components of a query can be shared with the other subgoals.

Concerning limitations, NAIF-resolution inherits from its "parents" the problem of the *dead end*, which represents the obstacle to the completeness. The dead end derives from infinite computations and the inability to compute bindings for variables, in some cases. However, *floundering* is avoided in all the cases in which it is possible to treat universal failure.

Obviously, the query language we obtain from this amalgamation is more expressive than the standard language of Horn clauses or general clauses, as a larger subset of the first order syntax is available for expressing queries. Our contention is that $\Gamma$*−general* queries are as expressive as extended queries, i.e. queries whose body can be any first-order formula ([11]). The only requirement for implementing such an extension is a sound form of the rule resulting by the amalgamation of NAF and NAI. The use of NAF and NAI rules results in an interpreter which is simpler to implement than other proposals such as intentional negation [2] or constructive negation [3].

In Section 3 we show correctness of NAIF-resolution w.r.t. the $Comp_L(P)$ semantics, for $\Gamma$*−general* programs and goals. Then, we give some examples showing the possible applications of $\Gamma$*−general* programs and how NAIF-resolution computes with them. In Section 4 we prove that NAIF-resolution is correct also for extended programs and goals, by first transforming them into $\Gamma$*−general* programs and goals, and then using the results in Section 3. Section 5 contains a comparison with the correctness results for extended programs and goals obtained by Lloyd and Topor in [11] for SLDNF-resolution.

# 2 Preliminaries

Theoretical foundations of the Negation As Failure rule and SLDNF-resolution can be found in [4],[1] and [10]. We refer to [6], [7] and [8] for background material on the Negation As Instantiation rule and SLDNI-resolution.

In the paper we will use the following notions and notations.

Through the paper we always refer to a language $L$ which contains infinitely many constant symbols.

Given a formula $Q$, we denote by $\mathit{freevar}(Q)$ the free variables in $Q$. The notation $T \models Q$

Table 1: The $\Gamma$–*general* language

| | |
|---|---|
| Queries | $Q ::= A \mid \exists\bar{x}.\neg(Q) \mid \forall\bar{x}.\neg(Q) \mid Q, Q$ |
| Goals | $G ::= \leftarrow Q$ |
| Clauses | $C ::= A \leftarrow Q \mid A \leftarrow .$ |

means that $Q$ is a logical consequence of the theory $T$; the symbol "," which might occur in $Q$ has to be interpreted as the logical conjunction "$\wedge$". We indicate a sequence of variables $x_1, \ldots, x_n$, (constants $d_1, \ldots, d_n$), with $\bar{x}$, ($\bar{d}$).

The formula $\Gamma\bar{x}.\neg(Q)$, where $\Gamma$ stands for $\exists$ or $\forall$, can leave some variables $\bar{y}$ free, which will be interpreted as suggested by the context. For the sake of simplicity, when it is not ambiguous, we write $\exists\bar{x}.\neg Q$, ($\forall\bar{x}.\neg Q$), and $\neg Q$ for $\exists\bar{x}.\neg(Q)$, ($\forall\bar{x}.\neg(Q)$), and $\neg(Q)$. Moreover, we use $\neg Q$ as an abbreviation for $\exists\emptyset.\neg Q$, and $\exists.\neg Q$, ($\forall.\neg Q$), for $\exists\bar{x}.\neg Q$, ($\forall\bar{x}.\neg Q$), with $\bar{x} = \mathit{freevar}(Q)$.

On the set of the atoms, the relation $A \leq A'$ ($A$ is more general than $A'$) holds iff there exists $\vartheta$ such that $A\vartheta = A'$. Associated to it is the equivalence relation (*renaming*) $\approx$. In other words, $A$ is a renaming of $A'$ ($A \approx A'$) if there exist $\vartheta$ and $\vartheta'$ such that $A\vartheta = A'$, and $A'\vartheta' = A$; $\vartheta$ ($\vartheta'$) is called a renaming for $A$ ($A'$).

For $V = \{x_1, \ldots, x_n\}$, $\vartheta$ a substitution and $p$ a predicate symbol, we define the predicate $\mathit{Inst}(\vartheta, V)$ as follows:

$$\mathit{Inst}(\vartheta, V) \Leftrightarrow p(x_1, \ldots, x_n)\vartheta \not\leq p(x_1, \ldots, x_n).$$

Intuitively, $\mathit{Inst}(\vartheta, V)$ is true if and only if $\vartheta$ instantiates $V$ and it is not a renaming.

# 3 NAIF-resolution

In this section we formalize the extension of SLD-resolution based on the treatment of negative information via both the NAI and the NAF rules. We consider programs which might have existentially or universally quantified negative queries in the bodies of the clauses, which we call $\Gamma$–*general* programs ($\Gamma$ standing for $\exists$ or $\forall$).

The syntax for $\Gamma$–*general* queries, $\Gamma$–*general* goals and $\Gamma$–*general* programs is described in Table 1, where $A$ is an atom.

The notions of literal and selection rule are extended as follows.

**Definition 3.1** Let $Q$ be a query. The *conjuncts* of $Q$, *Conj*(Q), is the smallest set such that

- if $Q \equiv A$ then *Conj*(Q)={A}
- if $Q \equiv \exists\bar{x}.\neg(Q')$ then $\mathit{Conj}(Q) = \{\exists\bar{x}.\neg(Q')\}$

- if $Q \equiv \forall\bar{x}.\neg(Q')$ then $Conj(Q) = \{\forall\bar{x}.\neg(Q')\}$
- if $Q \equiv Q_1, Q_2$ then $Conj(Q) = Conj(Q_1) \cup Conj(Q_2)$.

**Definition 3.2** A *selection rule* specifies, for every goal $\leftarrow Q$, one conjunct of $Q$ (the selected conjunct).

Roughly, the notion of derivation extends as follows. Let $P$ be a $\Gamma$*–general* program, $R$ a selection rule, and $\leftarrow Q$ the current ($\Gamma$*–general*) goal. Let $Q'$ be the selected conjunct. If $Q'$ is an atom, then we choose a clause $C$ of $P$ whose head unifies with $Q'$, and replace $Q'$ by the body of $C$, in $Q$. Finally, we apply the *mgu* to all variables of the resulting goal which are not in the scope of a (universal or existential) quantifier. If such a clause does not exist, then the derivation fails.

If $Q'$ is of the form $\exists\bar{x}.\neg(Q'')$, then we check whether the derivation tree for $P \cup \{\leftarrow Q''\}$ instantiates $\bar{x}$. If this is the case, then $Q'$ is removed from $\leftarrow Q$. If there exists a refutation for $P \cup \{\leftarrow Q''\}$ which does not instantiate the free variables of $Q''$, then the derivation fails.

If $Q'$ is of the form $\forall\bar{x}.\neg(Q'')$, then we check whether the derivation tree for $P \cup \{\leftarrow Q''\}$ is finitely failed. If this is the case, then $Q'$ is removed from $\leftarrow Q$. If there exists a refutation for $P \cup \{\leftarrow Q''\}$ which does not instantiate the free variables of $\forall\bar{x}.\neg(Q'')$, then the derivation fails.

The derivation succeeds if it ends with the empty goal.

There are also derivations which neither succeed nor fails. They can be infinite, like in $\{p \leftarrow p\} \cup \{\leftarrow p\}$, or loop forever in the attempt to solve a negative conjunct, like in $\{p \leftarrow \neg q, q \leftarrow \neg p\} \cup \{\leftarrow p\}$, or come to a *dead end.* This occurs when:

– the selected conjunct is $\exists\bar{x}.\neg Q$ and there are neither derivation trees which finitely instantiate $\bar{x}$, nor refutations non-instantiating the free variables of $Q$ for $P \cup \{\leftarrow Q\}$,

– the selected conjunct is $\forall\bar{x}.\neg Q$ and there are neither finitely failed derivation trees, nor refutations non-instantiating the free variables of $\forall\bar{x}.\neg Q$ for $P \cup \{\leftarrow Q\}$.

In these cases our interpreter must block because no answers are received about the subsidiary evaluation of $Q$ which possibly is infinite or fallen itself in a dead end.

Note that the test for success or failure of $\exists\bar{x}.\neg(Q'')$ could be equivalently done by checking whether $P \cup \{\leftarrow Q''[\bar{d}/\bar{x}]\}$ has a finitely failed derivation tree or a non-instantiating refutation, where $\bar{d}$ are fresh constant symbols.

We now give a formal definition of the notions of *NAIF-refutation* and *NAIF-tree.*

**Definition 3.3** Let $P$ be a $\Gamma$*–general* program. $R$ a selection rule, and $G$ a $\Gamma$*–general* goal.

- A *NAIF-tree of rank k* for $P \cup \{G\}$, via $R$, is a tree $TR$ satisfying the following:
  - ($a$) Every node is a $\Gamma$*–general* goal, every edge is labeled by a substitution,
  - ($b$) the root node is $G$,
  - ($c$) for every node $\leftarrow Q$
    - ($i$) if the selected conjunct is an atom $A$, then for each clause $H \leftarrow Q'$ in $P$ (standardized apart), if $H$ and $A$ are unifiable with mgu $\vartheta$, then the goal $\leftarrow ((Q \setminus A), Q')\vartheta$ is a child of $\leftarrow Q$, and the edge is labeled by $\vartheta$; if there are no such children the node is a (failed) leaf node;

(*ii*) if the selected conjunct is $\exists\bar{x}.\neg(Q')$ and there exists a finitely instantiating NAIF-tree for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$, then $\leftarrow Q \setminus \exists\neg(Q')$ is the only child of $\leftarrow Q$, and the edge is labeled by $\varepsilon$; if there exists a non-instantiating NAIF-refutation of rank $k' < k$ for $P \cup \{\leftarrow Q'\}$, then $\leftarrow Q$ is a (failed) leaf node.

(*iii*) if the selected conjunct is $\forall\bar{x}.\neg(Q')$ and there exists a finitely failed NAIF-tree for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$, then $\leftarrow Q \setminus \forall\bar{x}.\neg(Q')$ is the only child of $\leftarrow Q$, and the edge is labeled by $\varepsilon$; if there exists a NAIF-refutation of rank $k' < k$ for $P \cup \{\leftarrow Q'\}$ which does not instantiate $\mathit{freevar}(Q) \setminus \bar{x}$, then $\leftarrow Q$ is a (failed) leaf node.

- *TR finitely instantiates* $\bar{x} \subseteq \mathit{freevar}(Q)$ at level $h$ iff for each branch $\xi$ either

  (*i*) $\xi$ fails at a level $h' \leq h$, or

  (*ii*) $\mathit{Inst}(\vartheta_1 \cdots \vartheta_{k'}, \bar{x})$ is true, where $\vartheta_1, \ldots, \vartheta_{k'}$ label the edges of $\xi$ up to a level $h' \leq h$.

- *TR* is *finitely failed* at level $h$ iff its branches all fail at level $h$.

- A *NAIF-refutation of rank* $k$, $\xi$, for $P \cup \{G\}$ is a sequence of goals $\xi = G_0, G_1, \ldots, G_n$, with $G_0 = G$ and $G_n = \Box$, and an associated sequence of substitutions $\vartheta_1, \ldots, \vartheta_n$ such that for each $i \in [0, n-1]$, either

  (*i*) the selected conjunct in $G_i$ is an atom $A$, there exists a clause $H \leftarrow Q$ in $P$ (standardized apart) such that $H$ is unifiable with $A$ with *mgu* $\vartheta_{i+1}$ and $G_{i+1} = \leftarrow ((G_i \setminus A), Q)\vartheta_{i+1}$, or

  (*ii*) the selected conjunct in $G_i$ is $\exists\bar{x}.\neg(Q)$, there exists a NAIF-tree of rank $k' < k$ for $P \cup \{\leftarrow Q\}$ which finitely instantiates $\bar{x}$, $G_{i+1} = G_i \setminus \exists\bar{x}.\neg(Q)$ and $\vartheta_{i+1} = \varepsilon$.

  (*iii*) the selected conjunct in $G_i$ is $\forall\bar{x}.\neg(Q)$, there exists a finitely failed NAIF-tree of rank $k' < k$ for $P \cup \{\leftarrow Q\}$, $G_{i+1} = G_i \setminus \forall\bar{x}.\neg(Q)$ and $\vartheta_{i+1} = \varepsilon$.
  The substitution $(\vartheta_1 \cdots \vartheta_n)_{|G}$ is the *computed answer substitution.*

- $\xi$ is *non-instantiating* if $\mathit{Inst}(\vartheta_1 \cdots \vartheta_n, \mathit{freevar}(G))$ is false. In this case the computed answer substitution is $\varepsilon$.

## 3.1 Correctness (and completeness?) of NAIF-resolution

In this section we define the completion of a $\Gamma$*−general* program and we show that the NAIF-resolution is correct w.r.t. it.

In the next definition and theorems, the variables which occur in the formulas not explicitly quantified, have to be intended as universally quantified.

**Definition 3.4** Let $P$ be a $\Gamma$*−general* program. Its completion with respect to the extended language $L$ is defined in the usual way, namely $\mathit{Comp}_L(P) = \mathit{IFF}(P) \cup \mathit{CET}_L$, where $\mathit{IFF}(P)$ is the collection of completed definitions of predicates in $P$, i.e. the set of formulas

$$p(\bar{x}) \leftrightarrow \bigvee_{i \in [1,s]} E_i,$$

with $\bar{x} = x_1, \ldots, x_n$ and, for all $i$, if the $i$-th clause in $P$ with $p$ in the head is of the form $p(\bar{t_i}) \leftarrow Q_i$, then $E_i$ is the formula $\exists\bar{y_i}.(\bar{x} = \bar{t_i} \wedge Q_i)$, where $\bar{y_i}$ denotes the free variables in $p(\bar{t_i}) \leftarrow Q_i$.

**Theorem 3.5 (Correctness of NAIF-resolution)** *Let $P$ be a $\Gamma$−general program, $\leftarrow Q$ a $\Gamma$−general goal and $\bar{x} \subseteq freevar(Q)$. Then*

1. *if $P \cup \{\leftarrow Q\}$ has a NAIF-refutation with c.a.s. $\vartheta$, then $Comp_L(P) \models Q\vartheta$,*

2. *if $P \cup \{\leftarrow Q\}$ has a NAIF-tree which finitely instantiates $\bar{x}$ , then $Comp_L(P) \models \exists\bar{x}.\neg Q$.*

3. *if $P \cup \{\leftarrow Q\}$ has a finitely failed NAIF-tree, then $Comp_L(P) \models \forall\bar{x}.\neg Q$.*

**Proof** The proof is in part obtained by a slight modification of the proof of Theorem 4.7 in [8], which gives the correctness of SLDNI-resolution.

Concerning completeness, we already noted that NAIF-resolution suffers from the problem of the dead end. Obviously, to the purpose of approximating completeness as much as possible, it is better to delay the selection of a negative conjunct until all the positive conjuncts have been resolved, and generated all possible bindings.

**Example 3.6** Consider the program

$$\begin{aligned} P \;=\; \{ \; & p(x) \leftarrow \neg q(x,y), r(y), \\ & q(x,a) \leftarrow, \\ & r(a) \leftarrow . \qquad\qquad \} \end{aligned}$$

We have $Comp_L(P) \models \neg p(x)$, but, due to the local variable $y$, NAIF-resolution would not derive the success of $P \cup \{\leftarrow q(x,y)\}$ (it comes to a dead end), if we select $\neg q(x,y)$ before $r(y)$.

In case of positive programs and queries the dead end problem doesn't occur, hence completeness holds.

**Theorem 3.7** *Let $P$ be a positive program and $\leftarrow Q$ a positive goal.*

*If $Comp_L(P) \models \exists\bar{x}.\neg Q$, then there exists a NAIF-tree for $P \cup \{\leftarrow Q\}$ which instantiates $\bar{x}$.*

*If $Comp_L(P) \models \forall\bar{x}.\neg Q$, then there exists a finitely failed NAIF-tree for $P \cup \{\leftarrow Q\}$.*

**Proof** For the proof, note that in this case the NAIF-trees for $P \cup \{\leftarrow Q\}$ are just SLD-trees, then apply the completeness results for the NAI and NAF rules.

## 3.2 Examples

We give some examples showing how several problems can be naturally expressed in our language in a form very similar to their specification.

**Example 3.8** In logic programming sets are usually represented as lists, and the "membership" relation is defined as

$$\begin{aligned} & member(x, [x|y]) \leftarrow, \\ & member(x, [z|y]) \leftarrow member(x, y). \end{aligned}$$

Consider now the *subset* relation, formally expressed by

$$\begin{aligned} Y \subseteq Z \quad &\Leftrightarrow \quad \forall x.(x \in Y \Rightarrow x \in Z) \\ &\text{i.e.} \\ Y \subseteq Z \quad &\Leftrightarrow \quad \forall x.\neg(x \in Y \wedge x \notin Z) \end{aligned}$$

In our language this relation can be correctly defined as

$$subset(y, z) \leftarrow \forall x.\neg(member(x, y), \neg member(x, z)).$$

This definition is "complete" w.r.t. the positive use, in the sense that, given two ground lists $t, u$ which represent two sets $T, U$ with $T \subseteq U$, then the query $\leftarrow subset(t, u)$ has a NAIF-refutation. It is also complete w.r.t. the negative use, in the sense that if $T \not\subseteq U$, then the query $\leftarrow \neg subset(t, u)$ succeeds, because $\leftarrow subset(t, u)$ has a finitely failed NAIF-tree.

Note that if we eliminate "$\forall x$" from the definition, then it is not complete in the negative use anymore.

**Example 3.9** The *set-disjunction* relation, formally expressed by

$$\begin{aligned} Y \cap Z \quad &\Leftrightarrow \quad \forall x.(x \in Y \Rightarrow x \notin Z) \\ &\text{i.e.} \\ Y \cap Z \quad &\Leftrightarrow \quad \forall x.\neg(x \in Y \wedge x \in Z), \end{aligned}$$

in our language can be correctly defined as

$$disjunct(y, z) \leftarrow \forall x.\neg(member(x, y), member(x, z)).$$

One can easily verify that also this definition is complete in both the negative and the positive use.

**Example 3.10** Consider a binary operation *op*, defined in Logic Programming by a ternary relation $p_{op}$ (i.e. $p_{op}(x, y, z) \Leftrightarrow op(x, y) = z$). The *neutral element*, formally defined as

$$neutral(x) \Leftrightarrow op(x, y) = y \wedge op(y, x) = y$$

can be defined in our language by

$$neutral(x) \leftarrow \neg\exists y.\neg(p_{op}(x, y, y), p_{op}(y, x, y)).$$

The opposite can be defined as

$$not\_neutral(x) \leftarrow \neg neutral(x)$$

or, more simply

$$not\_neutral(x) \leftarrow \exists y.\neg p_{op}(x, y, y),$$
$$not\_neutral(x) \leftarrow \exists y.\neg p_{op}(y, x, y).$$

Observe the nested use of quantifiers in both Examples 3.8 and 3.10 of the form, respectively, $\forall \bar{x}.\neg(\ldots \exists \bar{y}.\neg(\ldots))$ with $\bar{y} = \emptyset$, and $\exists \bar{x}.\neg(\ldots \exists \bar{y}.\neg(\ldots))$ with $\bar{x} = \emptyset$. The next shows how more complicated relations as the Smyth preordering can be expressed by nesting the universal quantifiers.

**Example 3.11** The Smyth preordering relation is formally expressed by

$$Y \preceq_S Z \Leftrightarrow \forall x \in Z.\ \exists w \in Y.\ \ w \leq x$$

where "$\leq$" is a partial order on sets which suppose already defined. In our language we can define it by

$$Smyth(y, z) \leftarrow\ \forall x.\neg(member(x, z), \forall w.\neg(member(w, y), leq(w, x))).$$

**Example 3.12** This example shows the limits of NAIF-resolution deriving from its inability to compute bindings.

The *implication* predicate can be formally defined as

$$x \supset y \Leftrightarrow\ \ y \text{ is true whenever } x \text{ is true.}$$

A $\Gamma-$*general* program defining it could be:

$$\begin{aligned} P \ \ = \ \ \{ \ & Implies(x, y) \leftarrow \neg(true(x), false(y)), \\ & true(T) \leftarrow, \\ & false(F) \leftarrow . \qquad\qquad\qquad\qquad \} \end{aligned}$$

If we use the program $P$ to check whether $t \supset u$ holds for $t$ and $u$ being one of the truth values $T$ or $F$, then it is complete both in the positive and in the negative use. For instance, $\leftarrow Implies(T, T)$ succeeds, and $\leftarrow Implies(T, F)$ fails. However, if our aim is to find values $x$ and $y$ for which $Implies(x, y)$ is true (or false), then we don't obtain any answers because of the dead end.

# 4 Correctness for extended programs

From previous examples we can argue that $\Gamma-$*general* programs are more expressive than general programs, in the sense that they allow to derive programs from their logical specification quite easily. Extended programs, whose clauses can have as body any first-order formula, are even more expressive. For these programs Lloyd and Topor have shown that SLDNF-resolution is correct, by translating them into general programs ([11]). In a similar way, we show that NAIF-resolution can be correctly applied to extended programs previously transformed into $\Gamma-$*general* programs. This transformation is very simple, so the correctness result derives immediately from Theorem 3.

An *extended program* is a finite set of first-order formulas of the form $A \leftarrow W$, where $A$ is an atom and $W$ is a first-order formula.

An *extended goal* is a first-order formula of the form $\leftarrow W$, where $W$ is a (not necessarily closed) first-order formula.

Note that SLDNF and SLDNI-resolution only allow program clauses of the form $\leftarrow W$, where $W$ is respectively a conjunction of literals and a conjunction of $\exists\neg$*general* queries ([8]). We will see that every first-order formula is logically equivalent to a $\Gamma-$*general* query.

Next we give the definition of completion of an extended program $P$ w.r.t. the language.

**Definition 4.1** Let $P$ be an extended program. Its completion w.r.t. the language $L$ is defined in the usual way, namely $Comp_L(P) = IFF(P) \cup CET_L$, where $IFF(P)$ is the collection of completed definitions of predicates in $P$, i.e. the set of formulas

$$p(\bar{x}) \leftrightarrow \bigvee_{i \in [1,s]} E_i,$$

where $\bar{x} = x_1, \ldots, x_n$ are variables not appearing anywhere in the definition of $p$, and, for all $i$ s, if the $i$-th clause in $P$ with $p$ in the head is of the form $p(\bar{t}_i) \leftarrow W_i$, then $E_i$ is the formula $\exists \bar{y}_i.(\bar{x} = \bar{t}_i \wedge W_i)$, where $\bar{y}_i$ denotes the variables in $p(\bar{t}_i)$ and the free variables in $W_i$.

**Definition 4.2** Given two formulas $W_1$ and $W_2$ of a first order language $\mathcal{L}$ we say that $W_1$ is *logically equivalent* to $W_2$ if $\forall(W_1 \leftrightarrow W_2)$ is valid, i.e. it is true in every interpretation of $\mathcal{L}$.

**Proposition 4.3** *For each first-order formula $W$, there is a $\Gamma$–general query $Q$ such that $W$ and $Q$ are logically equivalent.*

**Proof** (By induction on the structure of $W$)

($W = A$) Obvious.

($W = \neg W'$) By inductive hypothesis there exists a $\Gamma$–*general* query $Q'$ such that $W'$ and $Q'$ are logically equivalent, then also $\neg W'$ and $\neg Q'$ are logically equivalent. Finally observe that $\neg Q'$ is equivalent to the $\Gamma$–*general* query $\exists \emptyset . \neg Q'$.

($W = W_1 \wedge W_2$) By inductive hypothesis there exist $\Gamma$–*general* queries $Q = Q_1, Q_2$ logically equivalent to $W_1$ and $W_2$. Then $Q \equiv Q_1, Q_2$ is a $\Gamma$–*general* query and it is still logically equivalent to $W$.

($W = W_1 \vee W_2$) Observe that $W$ is logically equivalent to $W' = \neg(\neg W_1 \wedge \neg W_2)$. Then apply the previous cases and the transitivity of the logical equivalence relation.

($W = W_1 \rightarrow W_2$) Observe that $W$ is logically equivalent to $W' = \neg(W_1 \wedge \neg W_2)$. Then apply the previous cases.

($W = W_1 \leftrightarrow W_2$) Observe that $W$ is logically equivalent to $W' = (W_1 \rightarrow W_2) \wedge (W_2 \rightarrow W_1)$. Then apply the previous cases and the transitivity of the logical equivalence relation.

($W = \exists W'$) By inductive hypothesis there exists a $\Gamma$–*general* query $Q'$ logically equivalent to $W'$. Now observe that the following is a sequence of logically equivalent formulas:

$$W, \neg(\neg(\exists W')), \neg(\forall \neg W'), \neg(\forall \neg Q'),$$

and that $\neg(\forall \neg Q')$ is a $\Gamma$–*general* query.

($W = \forall W'$) Observe that $W$ is logically equivalent to $W'' = \neg(\exists \neg W')$. Then apply the previous cases and the transitivity of the logical equivalence relation.

□

**Definition 4.4** Let $P$ be an extended program and $G$ an extended goal. The $\Gamma$–*general* forms of $P$ and $G$ are the $\Gamma$–*general* program $P'$ and the $\Gamma$–*general* goal $G'$ obtained by substituting in $P$ and in $G$ all the first-order formulas by the corresponding logically equivalent $\Gamma$–*general* query.

**Lemma 4.5** *Let $P$ be an extended program and $P'$ its $\Gamma$–general form. Then $P$ and $P'$ are logically equivalent and also $Comp_L(P)$ and $Comp_L(P')$ are logically equivalent.*

**Proof** Straightforward. □

**Lemma 4.6** *Let $P$ be an extended program, $P'$ its $\Gamma$–general form and $F$ a formula containing only predicates which appear in $P$. Then*

*If $Comp_L(P') \models F$ then $Comp_L(P) \models F$.*

**Proof** Obvious by Lemma 4.5 □

We now define the interpreter for extended programs. It is just NAIF-resolution as defined in Section 3, applied to the $\Gamma$–*general* forms of the extended programs.

**Definition 4.7** Let $P$ be an extended program, $G$ an extended goal and $R$ a selection rule. The NAIF-resolution for $P \cup \{G\}$ is the NAIF-resolution for $P' \cup \{G'\}$, where $P'$ and $G'$ are the $\Gamma$–*general* forms of $P$ and $G$.

**Theorem 4.8 (Correctness of NAIF-resolution for extended programs)** *Let $P$ be an extended program, $\leftarrow W$ an extended goal and $\bar{x} \subseteq freevar(W)$. Then*

1. *if $P \cup \{\leftarrow W\}$ has a NAIF-refutation with c.a.s. $\vartheta$, then $Comp_L(P) \models W\vartheta$,*
2. *if $P \cup \{\leftarrow W\}$ has a NAIF-tree which finitely instantiates $\bar{x}$ , then $Comp_L(P) \models \exists\bar{x}.\neg W$.*
3. *if $P \cup \{\leftarrow W\}$ has a finitely failed NAIF-tree, then $Comp_L(P) \models \forall\bar{x}.\neg W$.*

**Proof** By Theorem 3.5 and Lemma 4.6. □

# 5 Comparison with related works

The idea of using extended programs to solve general programming problems is not new. Clark himself, when introduced Negation As Failure, discussed about the transformation of extended programs into general programs (see [4] p.299), and a version of it has been implemented in micro-Prolog by Clark and McCabe, ([5]).

On the other hand, in [9] Kowalski asserted that " any problems can be converted from standard form to clausal form", (where standard form means the formalism using the full first order logic), and that "useful inference system for the standard form of logic may be obtained by combining inference rules for clausal form with rules for converting from standard to clausal form".

This is what Lloyd and Topor have done in [11], where a system of rules transforming an extended program $P$ into a general program $P'$ is given, and SLDNF-resolution is applied to the transformed programs. In this way, they prove correctness of SLDNF-resolution for extended programs and goals, justifying the introduction of them in logic programming by the fact that, for a given problem, an extended program is usually clearer and simpler than the corresponding general program because it is in a form similar to the (logical) specification of the problem.

The same reasons motivate our arrangement of NAIF-resolution to compute with extended programs. Anyway, our translation is more simple and immediate than the transformation used by Lloyd and Topor. We merely substitute verbatim in every extended clause $A \leftarrow W_1, \ldots, W_n$ all the formulas $W_i$ in a logically equivalent $\Gamma$*−general* query, thus obtaining a $\Gamma$*−general* program. No clauses are added to the programs. On the transformed program we use NAIF-resolution instead of SLDNF-resolution. This makes our system more powerful than the system based on SLDNF-resolution as explained in Section 1.

**Example 5.1** Consider the extended program

$$P = \{\ p(a) \leftarrow .\ \ \} \tag{1}$$

with the extended goal $G = \leftarrow \forall x.p(x)$.

The $\Gamma$*−general* form of $G$ is $G' = \leftarrow \neg\exists x.\neg p(x) \equiv \leftarrow Q$. We have that $P \cup \{G'\}$ fails in the NAIF system, and in fact $Comp_L(P) \models \neg Q$, i.e. $Comp_L(P) \models \exists x.\neg p(x)$. On the other hand, the translation of $G$ in a general form, as done by Lloyd and Topor is $G'' = \leftarrow answer()$, with

$$answer() \leftarrow \neg q(),$$

and

$$q() \leftarrow \neg p(x)$$

added to $P$ to obtain $P''$, and SLDNF-resolution flounders with $P'' \cup \{G''\}$.

**Example 5.2** Imagine a scene in which some blocks A, B, C and D are disposed in a way such that D is over B and C, which are over A. Consider the problem to show that the block A stands under all the others. The situation can be described by the following extended program and goal:

$$\begin{aligned} P &= \{\ Over(x, A) \leftarrow, \\ &\qquad Over(D, B) \leftarrow, \\ &\qquad Over(D, C) \leftarrow, \\ &\qquad Covered(y) \leftarrow \forall x.Over(x, y) \quad \} \\ G &= \leftarrow Covered(A). \end{aligned}$$

The $\Gamma$*−general* form of $P$ is

$$P' = (P \setminus \{Covered(y) \leftarrow \forall x.Over(x, y)\}) \cup \{Covered(y) \leftarrow \neg\exists x.\neg Over(x, y)\}.$$

We have that $P' \cup \{G\}$ succeeds in the NAIF system, and in fact $Comp_L(P) \models Covered(A)$. On the other hand, the translation of $P$ in a general program $P''$, as done by Lloyd and Topor, is obtained by substituting in $P$ the extended clause $Covered(y) \leftarrow \forall x.On(x, y)$ by the two general clauses $Covered(y) \leftarrow \neg p(y)$ and $p(y) \leftarrow \neg On(x, y)$, and SLDNF-resolution flounders with $P'' \cup \{G\}$.

Several examples of extended program clauses can be found in Kowalski's book (see Chapter 10 in [9]). We end this section by translating some of them to show the simplicity and immediacy of doing it.

**Example 5.3** ([9] p. 203) We want express that " Bob is happy if all his students like logic". The standard form is

$$Happy(Bob) \leftarrow \forall x.(Studentof(Bob, x) \rightarrow Likes(x, logic)).$$

The $\Gamma-$*general* form is

$$Happy(Bob) \leftarrow \forall x.\neg(Studentof(Bob, x), \neg Likes(x, logic)).$$

**Example 5.4** ([9], p. 203) A set is *well-founded* iff every non-empty subset has a least element. A set is non-empty iff it has at least one element. An element of a set is a least element iff it is less than or equal to every element of the set.

The standard form is

$$\begin{array}{rcl} Wellorderedset(x) & \leftrightarrow & \forall z.(Hasleastelmt(z) \leftarrow subset(z, x) \wedge Nonempty(z)) \\ Nonempty(z) & \leftrightarrow & \exists u.member(u, z) \\ Hasleastelemt(z) & \leftrightarrow & \exists u.[member(u, z) \wedge \forall v.(member(v, z) \rightarrow leq(u, v))]. \end{array}$$

The $\Gamma-$*general* form is

$$\begin{array}{rcl} Wellorderedset(x) & \leftarrow & \forall z.\neg(subset(z, x), Nonempty(z), \neg Hasleastelmt(z)) \\ Nonempty(z) & \leftarrow & member(u, z) \\ Hasleastelemt(z) & \leftarrow & member(u, z), \forall v.\neg((member(v, z), \neg leq(u, v))). \end{array}$$

Note that the completed definitions of $P$ are exactly the standard form given above.

**Example 5.5** Finally, we give the steps needed in the system of Lloyd and Topor to transform the extended clause defining the subset relation

$$C \equiv \; subset(y, z) \leftarrow \forall x.(member(x, y) \rightarrow member(x, z)).$$

into a general clause.

**step** 1. $subset(y, z) \leftarrow \forall x.\neg(member(x, y) \wedge \neg member(x, z))$.

**step** 2. $subset(y, z) \leftarrow \neg\exists x.(member(x, y) \wedge \neg member(x, z))$.

**step** 3. $subset(y, z) \leftarrow \neg p(y, z)$

and

$p(y, z) \leftarrow \exists x.(member(x, y) \wedge \neg member(x, z))$.

**step** 4. $subset(y, z) \leftarrow \neg p(y, z)$

and

$p(y, z) \leftarrow (member(x, y), \neg member(x, z))$.

Observe that in our system only step 1 is required to obtain the $\Gamma-$*general* form of $C$ (see Example 3.8), provided that "$\wedge$" is changed in ",".

# 6 Conclusions

We have introduced NAIF-resolution which is an amalgamation of SLDNF-resolution and SLDNI-resolution able to treat with quantified negative queries ($\Gamma-$*general* queries). We have shown that $\Gamma-$*general* programs are as expressive as extended programs by exhibiting a translation of the latter into the former. Then, we have proved correctness of NAIF-resolution for extended programs by first transforming them into $\Gamma-$*general* programs and then using the fact that NAIF-resolution is correct in this case.

Our translation is more immediate than the mechanism introduced by Lloyd and Topor to translate extended programs into general ones.

Moreover, we have shown that our system, which is based on NAIF-resolution, is more "complete" than the one of Lloyd and Topor which is based on SLDNF-resolution, i.e. it can infer more logical consequences of the completion semantics.

# References


[1] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] R. Barbuti, P.Mancarella, D. Pedreschi, and F. Turini. A trasformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[3] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, Cambridge, Mass., 1988.

[4] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[5] K.L. Clark and F. G. McCabe. *micro-PROLOG:Programming in Logic.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[6] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as Instantiation: a new rule for the treatment of negation in Logic Programming. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 32–45. The MIT Press, Cambridge, Mass., 1991.

[7] A. Di Pierro, M. Martelli, and C. Palamidessi. SLDNI-resolution: a non-floundering interpreter for geneneral programs. In P. Asirelli, editor, *Proc. Sixth Italian Conference on Logic Programming*, pages 293–307. The MIT Press, Cambridge, Mass., 1991.

[8] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation As Instantiation. Submitted for publication to *Information&Computation*, 1992.

[9] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[10] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[11] J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.